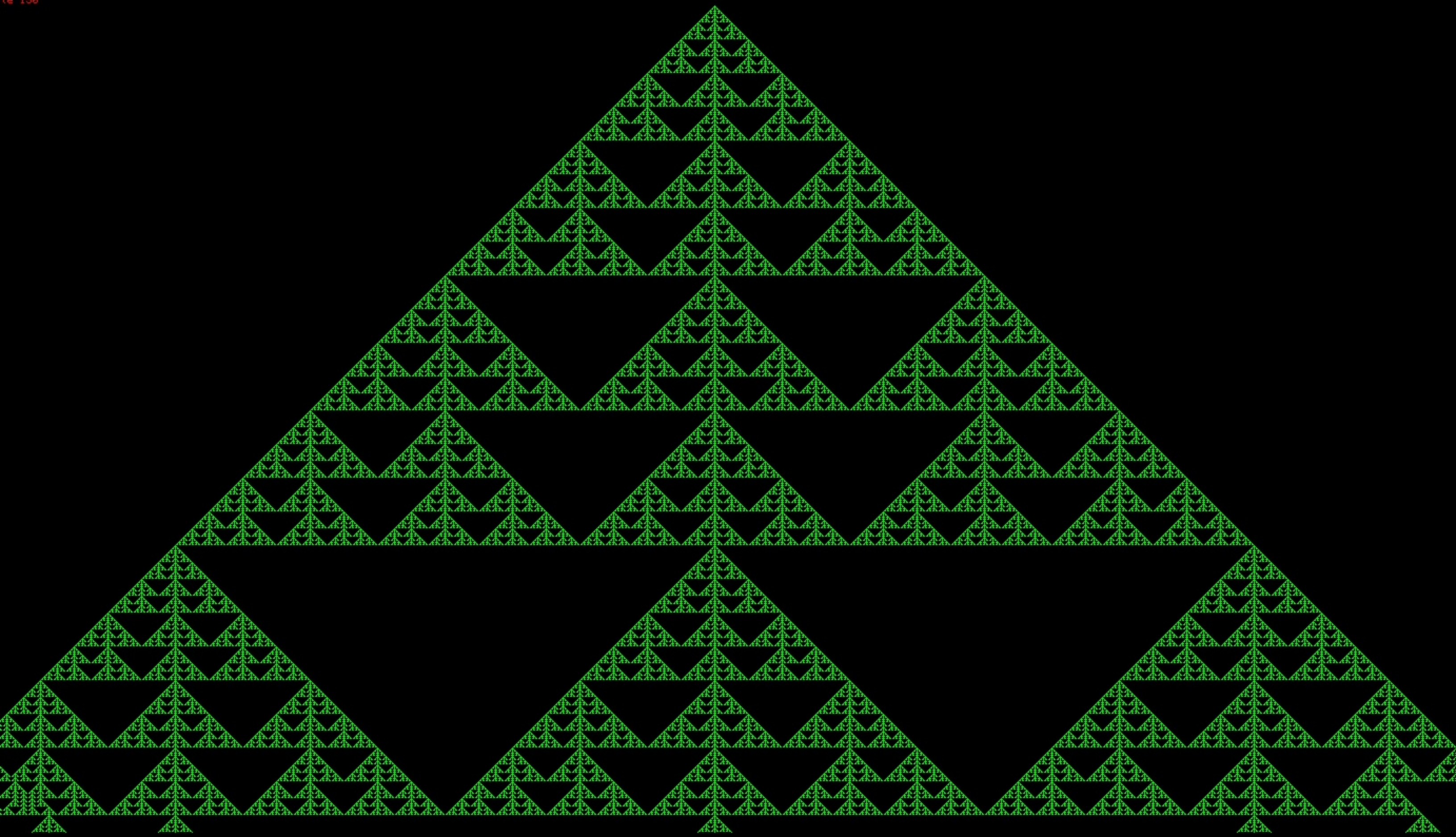
smsqzine

Issue #4 **March 2017**



SMSQzine

Table of Contents

Published by:
Timothy Swenson
swenson_t@sbcglobal.net
swensont@lanset.com

SMSQzine is published as a service to the Sinclair QL community. Writers are invited to submit articles for publication. Readers are invited to submit article ideas.

Created using Open Source Tools:

- OpenOffice
- Scribus
- Gimp
- SMSQmulator

Copyright 2017
Timothy Swenson

Creative Commons License
- Attribution
- Non-Commercial
- Share-Alike

You are free:

- To copy, distribute, display, and perform the work.

- To make derivitive works.

- To redistribute the work.

Editorial 1

Digital Precision C Front End 1

Orbit 2

Array Search 3

New Release of uQLx 3

Wander 4

Playing with Digital Precision C 6

Editorial

It's been a few months, but I have had enough projects and ideas to write articles for another issue of SMSQzine. The articles detail the different projects that I have been working. Some programing projects, some testing projects, some entertaining and some that are hopefully interesting. Even if you don't use any of the code mentioned in some of the articles, I hope that it gives you an idea for a project or code of your own.

Digital Precision C Front End

If I am writing a C program that uses QDOS calls, I find that Digital Precision C is my compiler of choice. I started years ago with the Small-C compiler, which eventually became Digital Precision C, so it is also the compiler that I am most familiar with.

```
        DP C Front End
Enter Device: (win1_) win2_
Enter Source: (no _c) window1

EXEC cc;win2_window1 -p -d/ram1_
EXEC cg;win2_window1 window1 -nc -p -d/ram1_
 1 = cc   2 = cg   3 = exit
```

The one problem with the compiler is that it is not able to handle directories. It expects all library and other files to be on the same device, be it WIN1_, FLP1_, or RAM1_. If given a path for source files, the compiler will use it, but there are some files where it just uses the default device.

In the old days of floppies, what I could normally do is copy everything to RAM1_ and run all compilation from there and then save to floppy when I was done. Now I have the compiler on WIN1_ and the source code on WIN2_. Since I have sub-directories on WIN1_, I can't use WIN1_ as my default device, so I've fallen back to using RAM1_ for storing the files the compiler is expecting.

Since I don't use the compiler all that often, it can take a few minutes to remember the syntax of the compiler (cc) and the code generator (cg) command lines. I thought it was about time to create a front-end for the compiler that will make it much easier to use.

The front end is just a short SuperBasic program that I have compiled with Turbo. It first copies the compiler necessary files to RAM1_. It gets from the user the device / directory of the source files (like WIN2_ or WIN2_src_). It then gets the name of the program to compile (without the _c extension). It constructs the proper command line for both the compiler and code generator and displays that on the screen.

The program then waits for input. If the 1 key is hit, then the compiler will be executed. If the 2 key is hit, then the code generator will be run. Hitting 3 exits the front end. When either phase of the compiler is run, it can execute and quit before there is time to it to display on the screen. To confirm that a key has been hit and acted on, a beep will sound.

With the front end being compiled, during a compile session it need only be executed once. Once editing of the source program is done and saved, CTRL-C to the front end, hit the keys to start the compiler and code generator. If there are errors, CTRL-C back to the editor and continue working on the source code. This is the way that front end is designed. If executed each time the compiler is needed to be run, it will query the user about copying the files over the existing files in RAM1_.

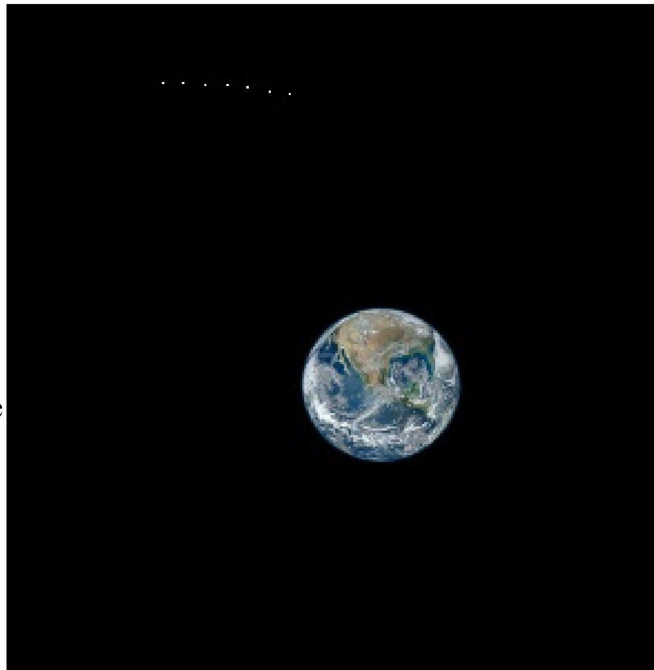
The front end is designed for my environment, using the directories that I use. It assumes that the library files are in WIN1_DPC_ directory. It is fairly easy to edit the code to adjust for a different environment and then fairly easy to compile with Turbo. With Turbo, I only changed the Task Name and the name of the output file. The rest were all just the defaults.

Orbit

A while back there was some discussion on the QL Forums chat about playing Kerbal Space Program, which is a game that lets one built rockets, launch them and orbit around another planet. It would be nice to try something like this on the QL, but Kerbal is way too complicated.

Opening screen for the game

I was able to dust off an old program and try for something similar, but much easier to program. Back in college I had "stolen" a program from a friend. The program is a simulation of an object orbiting around a planet. The program allows input from the user to add thrust to the object to affect the orbit. If the object is traveling to the left and the user adds right thrust, this will decelerate the object and let to move closer to the planet.



Satellite orbiting around Earth

I changed the program to have the starting point not be a a stable orbit. The user must add thrust to the object to get the right velocity at a certain location in space, to be in a stable orbit. The display shows the X and Y velocity of the object and the X and Y location. This helps the user hit a certain point at a certain velocity. Advanced players can tinker more with using thrust and see about moving from a circular orbit to a more elliptical orbit, or change the orbit to be close to the planet at one point and far from the planet at another.

To make the game more interesting, I decided to use some nice graphics and utilize the GD2 colors. I wanted to create a nice image of a planet. At first I tried just drawing a circle and then adding some additional color to it. Then I wanted to see if I could just copy a JPEG or PIC image to the screen.

David Westbury wrote a package called fun.zip and one of the extensions in that package is called FPIC_LOAD. This will take a _PIC file and load it to a location on the screen. My next step was to find an image that I could use. I found an image and tried to convert it from BMP to PIC and could not get that to work. Bob Spelton stepped up to help me and was able to send me the images in _PIC format.

As I was testing the program in SMSQmulator, I found that the images were only displaying about every other part of the image in stripes. I sent a query off to Wolfgang Lenerz and he sent me a beta release of the next version of SMSQmulator and that fixed the problem.

The game is designed to be run with a large screen, so you will need either QPC2 or SMSQmulator with a screen size of 1400 x 800. Both emulators can handle this resolution and they both support GD2 colors.

The game comes the toolkit needed for the game, a font, and a user guide in PDF. The game is written in SuperBasic and is not compiled. With SMSQmulator, I found that the game was fast enough and did not need any compilation.

Array Search

I am working on a project where I need to quickly find a word in a large array of words. My concern was how long it would take for each search and ways to make the search quicker. The project is to take a text file of a Structured SuperBasic program, find all of the SuperBasic keywords and capitalize them.

Before the search of the array is possible, the array has to be filled with a list of words. The word list is generated and stored in a text file. When the program is run, the text file is read into the array. The keyword file was generated by sending the output of the EXTRAS command to a file. Using SMSQmulator and having a number of toolkits loaded, the keyword list is 768 words long. The keyword list was also sorted to make things easier (more about this later). Since the sort needs to only be done once, there is no processing penalty for doing it.

The first method of searching is the brute force method, where the search starts at the top of the array and searches down through the array until the word is found or not found. It is a very simple search, but not very efficient. It has no requirements on how the data is arranged in the array. If the array is sorted or not, the search still works the same.

The second method is to handle the search a little more intelligently. With the keywords pre-sorted when they go into the array, all of the keywords that start with the same letter are grouped together. If a word that I am searching for starts with the letter D, then I only need to start searching in the array where the keywords that start with the letter D are located. Instead of searching all 768 words, in the worse case, the search only needs to search through the keywords that start with D.

As the keywords are read into the array, the character code of the first letter is tracked. If the character code changes, then this is the start of a different letter. Where in the array that character change takes place is stored in another array (indexed by the character code).

When given the search word, the first code of its first character is determined and used to look in the index of the first keyword in the array that begins with the same character. The search will only need to search the 10-30 or so words that begin with that character instead of the full array.

The second search will be faster, but by how much was unknown. To determine how much faster, I captured the system time when the search started and the system time when the search ended. QL DATE function will only go down to the second, so I had to run both searches 3,000 times to get any sort of valuable timing numbers.

The brute force search took 35 seconds for 3,000 searches. The index search took just about 1 second for 3,000 searches. The second search was not all that difficult to code, but it sure made for a much faster search.

The source code is included in the zip file for this issue.

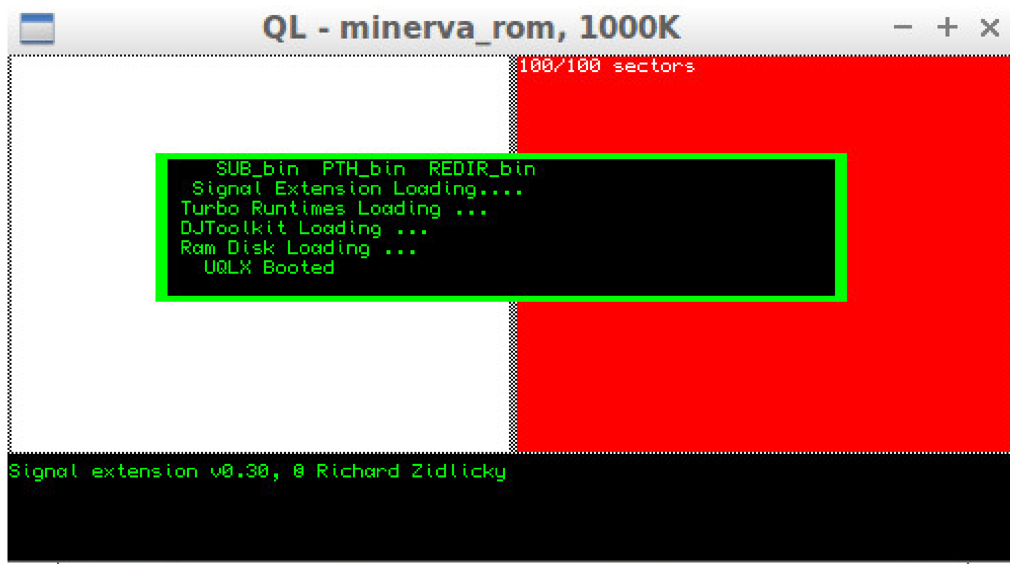
New Release of UQLX

Back in the late 1990's, Richard Zidlicky created uQLx, the QL emulator for Unix. I ran it on a SGI Indy workstation for a number of years, while I was working at SGI. It was not much longer after 2001 that Richard stopped working on uQLx and the code sat dormant.

At the "QL is 30" meeting, Graeme Gregory was talking with someone about uQLx and they said something about how it was impossible or real difficult to convert uQLx to run on 64-bit systems. Being a C coder by trade, Graeme thought this would be a good challenge and got a hold of the uQLx source to see what he could do.

After some serious cropping of some of the uQLx code, Graeme was able to get it to compile with 64-bit. He did have to go through the code looking for places that were set for 32-bit and fix a few other issues. Eventually he had a working version of uQLx.

UQLx has been tested with a number of QL programs. A number worked, and some did not. Since the main purpose of Graeme's work was to make uQLx compilable on newer systems, there was little effort put into fixing the main emulation core of uQLx. There are some programs that have never run on uQLx and probably will not run on the current version.



Boot screen for uQLx

The version that Graeme worked on is available from Github. You just need to install git on a Linux system, pull down the source code and compile it with gcc. To make it easier for users, I have put together a binary distribution of uQLx. I compiled the 32-bit version, Graeme compiled the 64-bit and he also compiled versions for ARM. Rob Heaton did the testing of the ARM versions.

I also put together a small QXL.win file to go with uQLx that has a few programs including unzip so that the user is able to use unzip to access a number of programs available for download. I also created a User Guide that walks the user through setting up the binary distribution of uQLx and also how access the source code on Gitbhub and compile it.

The binary release is now available on Dilwyn's web site on the emulation page:

http://www.dilwyn.me.uk/emu/index.html

Wander

In 1973, Gegory Yob wrote the game "Hunt the Wumpus", which was a simple game of going through a number rooms in a cave looking to kill the Wumpus before he kills you. The commands were simple in that you had a selection of connecting rooms to move into next. The rooms could have hazards in them that could harm you. If you thought the Wumpus was in the next cave room, you could shoot an arrow in the room to kill him. This was the first cave crawling game.

In Wikipedia, the article on "Interactive Fiction", has this statement:

"Around 1975, Will Crowther, a programmer and an amateur caver, wrote the first text adventure game. Adventure.."

This game later became known as Colossal Cave. It was written in Fortran for the PDP-10. The original game was based on actually crawling through a real cave. Don Woods was the one that added the fantasy elements to the game. In 1978, Scott Adams of Adventure International, wrote "Adventureland" based on Colossal Cave. first written for the TRS-80 Model 1.

For many years everyone through that Adventure by Will Crowther was the first text adventure game, later to be known as Interactive Fiction.

In 1973, Peter Langston wrote a program called Wander in BASIC. It was the first text adventure game, coming even before Colossal Cave. In 1974, Peter ported the game to C and to a "mainframe" environment. The game was mentioned in the Inform Designer's Manual and in a number of other places, but most thought the game was lost.

Recently, Peter dug through an old e-mail archive and was able to retrieve a slightly later version of the program, from about 1980.

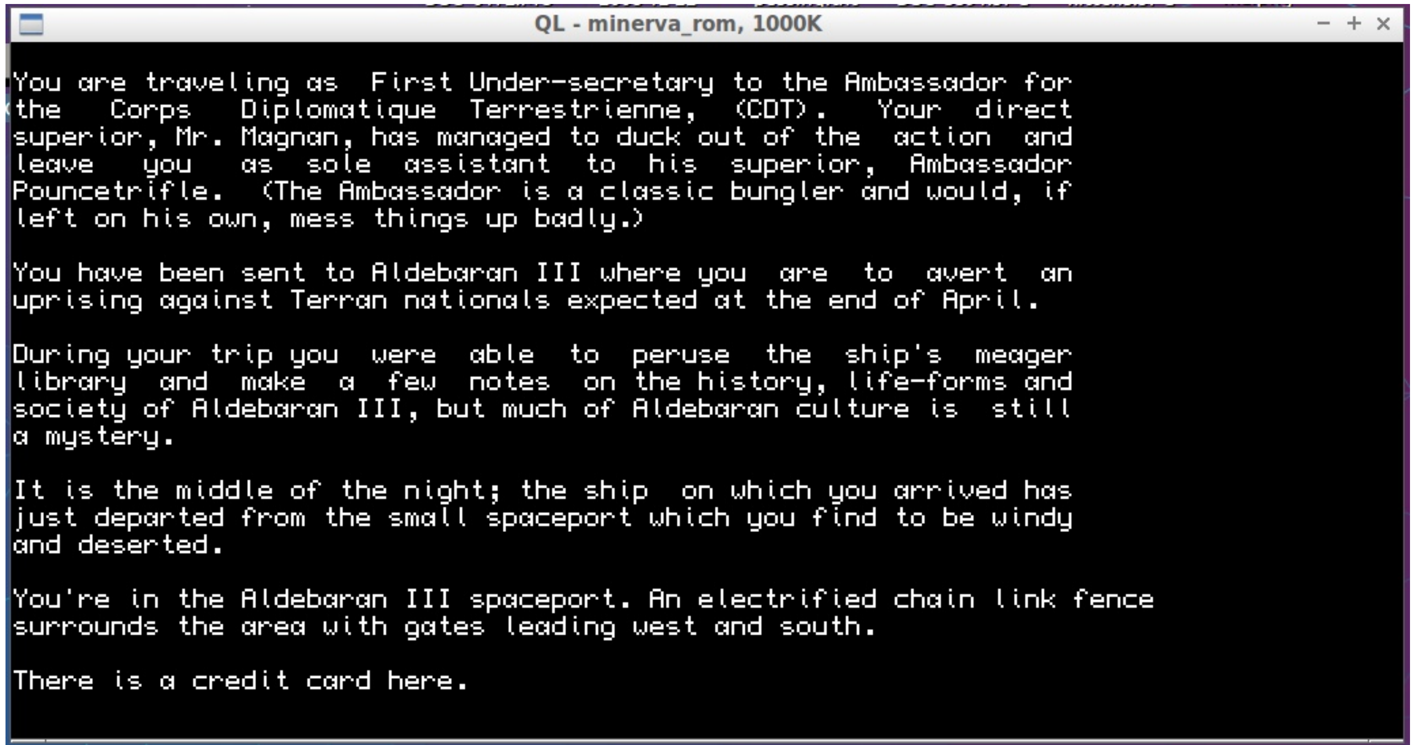
Wander is not just a text adventure game, but a game system. A single adventure is two files which Wander, the game engine, processes to run the adventure. Part of the distribution was the document that has the instructions on how to write the adventures and create the files that drive the game.

Wander comes with three adventures, the first starts at the Aldebaran III Spaceport where you are a diplomat trying to avert an uprising. In the second one, "Castle", you are taken back in time to a castle. The third one, is a tutorial to show how Wander works by having an adventure with binary math.

Each adventure is composed of a .misc and a .wrld file. The Misc (.misc) file contains the location dependent information and the World (.wrld) file contains location - state information. There are two text files that describe the contents and format of each of these files.

I downloaded the version that was made available and compiled it under Linux. It compiled with no changes and just ran. It tried to compile it on C68, but there were a number of errors. Not being a C expert, I passed Wander on to Graeme Gregory, our resident C expert, and was he able to compile it under QDOS-GCC compiler and get it working.

I'm not a fan of text adventure games, but given the recent discussion on the "Sinclair QL Forum", when I ran across the mention of Wander on the Slashdot website, I had to see if it was possible to get it running on the QL. Given that the QL is over 30 years old, I thought it would be interesting to run



Opening screen for the default adventure for Wander

software on the QL that was written years before the QL was even thought of.

There is an issue of Wander not working with the Signal Extensions, so don't load the sigext30_rext file before running Wander.

Wander is available on Dilwyn Jones website:

http://www.dilwyn.me.uk/games/adventures/wander.zip

Playing with Digital Precision C

Digital Precision C (DP C) is an older compiler that has not been updated in a long time, where as SMSQ/E has had a number of improvements over the years. I was wondering lately if DP C would be able to use any of the newer features of SMSQ/E.

Colors

With SMSQmulator, there are four color modes:

colour_QL	- 8 primary colors (64 w/ stipple)
colour_PAL	- 256 colors
colour_24	- 24-bit colors (16 million)

DP C was designed for the BBQL, which only had mode 4 and mode 8. I wondered if it could access more than the standard QL colors. The paper and ink system calls use INT (8 bits), so I knew that 24-bit color was not possible. So with 8-bits, it might be possible to do colour_PAL. I wrote a short SuperBasic program that set the paper to a color, cleared the screen and did this 256 times. I set the color mode and ran the program. This gave me my baseline for colors.

I wrote a similar C program. I made sure not to set the MODE in the program, hoping that the colors that I set before running the C program would come into play. I set the color to colour_QL and ran my C program. It got the 8 primary colors. I then set the color to colour_PAL, ran the program and still got the 8 primary colors. I also set the MODE in the C program to 4 and to 8, and I still got the same 8 primary colors. No matter the mode or color setting, DP C will only generate colour_QL colors.

Window Size

With larger window sizes, I wondered if DP C could create a window larger than the standard 512x256. I opened a large format window with SMSQmulator, then changed the code in my color program to open a window with dimensions of 1000x512. I ran the program and it generated the right size window. It looks like DP C can handle the larger window sizes.

Task Name

Both Turbo and C68 compilers let you set the name of the executable (task name) when listed in the jobs table. With DP C, all that is displayed in the jobs table is "Digital C". When writing programs, one might want the task name to be related to the program versus some generic name for all DP C programs. I checked the DP C manual and there is no way of setting the task name. I used a file hex editor and found where in the executable the task name is stored. I knew that it might be bad to change the size of the executable file, so if I was going to edit the file and change the task name, I had to not change the size of the name. With "Digital C", I had 9 characters to utilize. If I had a short task name, I could set the rest of the characters to the space character. The file editor worked great in editing the executable. Granted it is a little cumbersome to edit it by hand, but it only needs to be edited once, when the program is considered complete.

Console Window

With DP C, there is a default console window that is created by the compiler runtime. It is possible to override this default window by creating a function called _console(), where a new window is defined. As I was writing my test programs, I made a few coding errors when creating _console() and had to refer back to some older code about creating it.

Part of the _console() function is to open a window with a file pointer to hold the window pointer. In SuperBasic this would be "open #3" where 3 is stored in a variable. When the paper or ink is set on the window, the function call is made with the variable for the window like this:

```
      ink(fd);
      paper(fd);
```

The mistake I made was thinking that when I would right text to the window, I would use fprintf() like this:

```
      fprintf(fd,"Hello world\n");
```

But that did not work. Just using regular printf() was the answer. This was the way that I had written programs for years, but thinking about it some more, I realized that it was not the right way. I found an other DP C (or Small C) program by a different programmer and he had the right approach. I confirmed it by re-reading the DP C manual. What I find interesting is given how my code was wrong, it still did the job of opening the console window, setting the paper and ink colors. Since printf() is designed for STDOUT, I think that the new window sort of became the new STDOUT.

Below is my original code (window1_c) and the proper code (window2_c). Both program do basically the same thing, but each one takes a different approach to it. From now on, I'm going to use the second, more proper, way of writing the _console() function.

```
/* window1_c */

#include <stdio_h>

char  *fd;

_console() {
   fopen(fd,"con","w");
   window(fd,300,100,75,75);
   paper(fd,0);
   ink(fd,4);
   border(fd,2,2);
   cls(fd);
}

main()
{
   printf("This is a test.\n");
   pause(200);
}
```

```
/* window2_c */

#include <stdio_h>

_console() {
   int fd;

   fd =
fopen("con_300x100a75x75","w");
   paper(fd,0);
   ink(fd,4);
   border(fd,2,2);
   cls(fd);
   return(fd);
}

main()
{
   int fd;

   fd = _console();

   fprintf(fd,"This is a test.\n");
   pause(200);
}
```